Lab 6
LED Array Device Driver

EELE 468
SoC FPGAs II: Application Specific Computing

Due date: 04/26/2022

In this lab, we'll create a device driver for the LED array on the DE10 Nano. We're going to create the device driver from scratch, building it up bit-by-bit and learning about each and every piece along the way.

Quartus Project

We've provided a Quartus project for you to use. It is in the `quartus` directory in your Lab 6 GitHub repository. This project has a memory-mapped component for the DE10 Nano LED array; this component is located in the `led-array` directory.

You'll need to compile this Quartus project. Once the project is compiled, convert the `.sof` file into an `.rbf` file (passive parallel x16). Put the `.rbf` file onto your SD card so U-Boot programs your board with the new bitstream (`.rbf` file).

Introduction to Device Drivers

A good overview of the types of device drivers found in Linux can be seen in figure 1, which is from the book *Linux Device Drivers* found at https://lwn.net/Kernel/LDD3/.

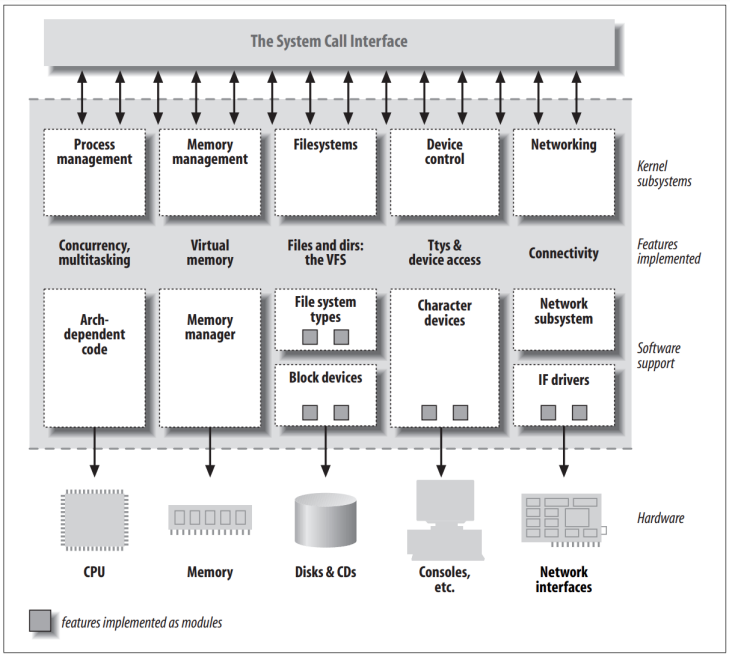

Figure 1-1. A split view of the kernel

Figure 1: **Types of Devices and Drivers found in Linux.** The three main classes of device drivers are *block devices*, *character devices*, and *network interfaces*.

Network interfaces are a special category that deal with the transmission and receipt of data packets from other computers.

Block devices are closely related to file systems where data movement is more efficient when the data is moved in blocks of 512 bytes or larger (and a power of two). Typically there is a memory buffer for storing blocks and the data can be accessed in random order (e.g. file seek).

Character devices involve transactions of single bytes (or single 32-bit words in our case) that are sequentially accessed (a stream of bytes read from a keyboard or sent to a serial port). Since we will be reading and writing single 32-bit data words to specific memory locations, the device driver type that we will implement will be a *character device driver* in order to control our custom device (i.e. reading and writing 32-bit words from/to our custom registers in our custom dataplane component in Platform Designer).

Buses

Desktop computers have two convenient buses, the *Universal Serial Bus* (**USB**) and the *Peripheral Component Interconnect* (**PCI** or **PCIe**) buses (e stands for express). Both of these buses allow hardware to be attached to the computer without having to inform the operating system (OS) ahead of time. These buses allow hardware to be **discoverable** where the hardware can be plugged in and the hardware says to the OS "Here I am" and tells the OS what device they are and what resources they have. The OS can then deal with them as they appear or disappear.

However, many times in embedded systems, there is **hardware that is not discoverable**. The OS needs to be informed that these hardware devices exist and what resources are available for them. These devices are regarded as connected to a *virtual bus*, which is called the **platform bus**. Linux device drivers that are created for these devices are known as **platform drivers**. Since our LED array component in Platform Designer is not discoverable, we will treat it has a platform device and create a platform driver for it.

In addition to the platform bus, the i2c and spi physical buses are commonly found in embedded systems. These devices aren't hot-pluggable and have to be defined statically, e.g. in the device tree. The kernel has subsystems for these busses, amongst many others that you might encounter in your lifetime as an embedded systems engineer.

Subsystems

The kernel has a large number of subsystems that cover a large range of device types, e.g. input, leds, network, audio, etc. These subsystems make writing drivers easier because they contain common functionality that all drivers of a particular type would need or want; they also help provide a consistent API to user-space, e.g. all keyboards will expose the same interfaces. Figure 2 shows some subsystems that utilize the character driver API.

While there are many subsystems in the kernel, some devices, e.g. custom devices in the FPGA fabric, don't fit cleanly into any of the subsystems. The **misc subsystem** was created to handle devices that don't cleanly fit into an existing subsystem. If the driver can make use of the *character driver* API and could be implemented as a *raw character driver*, the misc subsystem layers on top and makes the work of developing a character driver easier. We'll be using the misc subsystem in our driver.

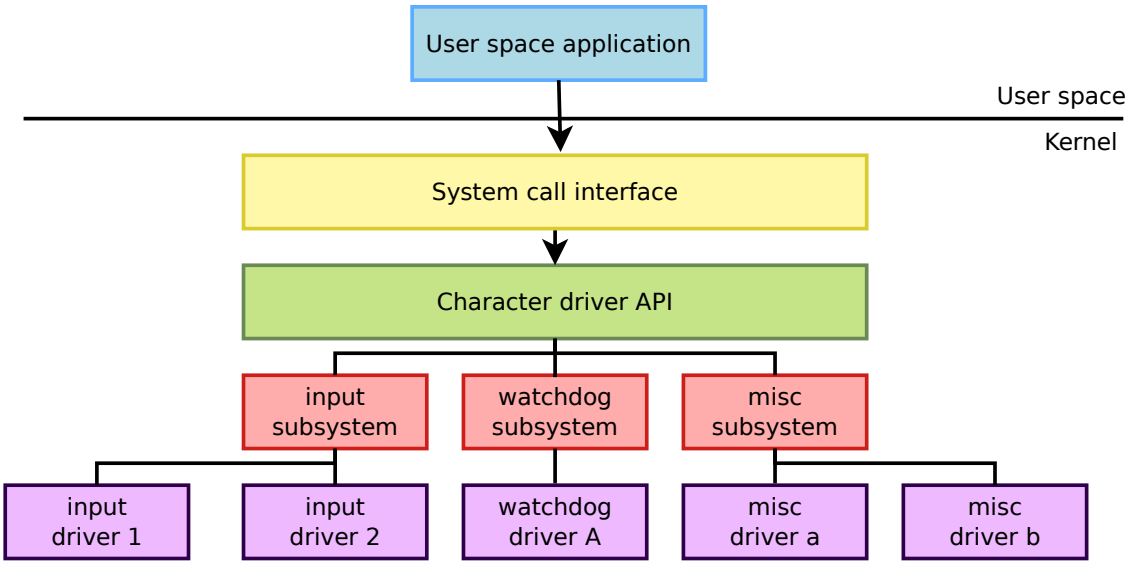

Figure 2: Diagram of different subsystems that use the character driver API. This diagram was created by engineers at Bootlin and is licensed under CC BY-SA 3.0.

Creating the Platform Device Driver

In order to better understand the driver, we will build it piece-by-piece. You'll have to make platform drivers for your final project, so pay attention! 🙂

Probing and Removing a Platform Device

We'll start by exploring how our driver inserts and removes platform devices. This exploration will be at a somewhat superficial level, since most of the details are handled by the kernel's driver core.

When the kernel is made aware of a device, it first has to find what driver(s) are compatible with the device. In our case, the device tree subsystem is what informs the kernel that our device exists. **If the `compatible` property in a device tree node matches the compatible string that we define in our driver, then the device gets bound to our driver.** Once a device has been bound to the driver, the driver's `probe` function gets called; similarly, when a device is removed from the system, the driver's `remove` function is called.

Before we can start probing and removing devices, we need to define our platform driver. Listing 1 shows the `plaform_driver` `struct` definition, and listing 2 shows what our platform driver instance will look like. We need to provide *function pointers* to our `probe` and `remove` functions, as well as the driver owner, name, and device tree match table (`of_match_table`)[1]. We also need to set up some module documentation macros; in particular, we need to use a GPL-compatible license because some of the platform driver code we use is exported with `EXPORT_SYMBOL_GPL()`, essentially meaning that our driver is a derived work of the kernel and must be licensed under a GPL-compatible license.

[1] `of` stands for Open Firmware, which is the standard that originally defined the device tree; see this page for a brief history

```c
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
        bool prevent_deferred_probe;
};
```

Listing 1: `platform_driver` **struct** definition from include/linux/platform_device.h. For our purposes, we only need to define the probe/remove function pointers and some fields in the `device_driver` **struct**, as shown in Listing 2. Look at the `device_driver` **struct** documentation here: include/linux/device/driver.h

```c
/**
 * struct led_array_driver - Platform driver struct for the led array driver
 * @probe: Function that's called when a device is found
 * @remove: Function that's called when a device is removed
 * @driver.owner: Which module owns this driver
 * @driver.name: Name of the led array driver
 * @driver.of_match_table: Device tree match table
 */
static struct platform_driver led_array_driver = {
        .probe = led_array_probe,
        .remove = led_array_remove,
        .driver = {
                .owner = THIS_MODULE,
                .name = "led-array",
                .of_match_table = led_array_of_match,
        },
};

/*
 * We don't need to do anything special in module init/exit.
 * This macro automatically handles module init/exit.
 */
module_platform_driver(led_array_driver);

MODULE_LICENSE("Dual MIT/GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("LED array driver");
MODULE_VERSION("1.0");
```

Listing 2: Our LED array platform driver **struct**. The `probe` and `remove` fields are function pointers to our probe and remove functions (defined in Listing 3). The `owner` field specifies which module owns the driver. `of_match_table` is what gets used to match device tree nodes to our driver. The `module_platform_driver()` macro creates the module init/exit code for us.

For now, we'll create trivial probe and remove functions that just let us know the function has been entered. Listing 3 shows the probe and remove functions you need to implement. Remember that you can get the *address* of a function by using the function name without parentheses, which is what is done in Listing 2.

```
/**
 * led_array_probe() - Initialize device when a match is found
 * @pdev: Platform device structure associated with our led array device;
 *        pdev is automatically created by the driver core based upon our
 *        led array device tree node.
 *
 * When a device that is compatible with this led array driver is found, the
 * driver's probe function is called. This probe function gets called by the
 * kernel when an led_array device is found in the device tree.
 */
static int led_array_probe(struct platform_device *pdev)
{
        pr_info("led_array_probe\n");

        return 0;
}

/**
 * led_array_probe() - Remove an led array device.
 * @pdev: Platform device structure associated with our led array device.
 *
 * This function is called when an led array devicee is removed or
 * the driver is removed.
 */
static int led_array_remove(struct platform_device *pdev)
{
        pr_info("led_array_remove\n");

        return 0;
}
```

Listing 3: Trivial probe and remove functions.

In order for our driver to be matched with compatible devices, we need to create an array of `of_device_id` **`struct`**s, as shown in listing 4. The `of_device_id` array should be **`static`** (not accessible by other files, i.e. private) and **`const`** (code shouldn't be able to modify the compatible string at run-time). **You need to set the compatible string to** `"lastname,led-array"`, **where** `"lastname"` **is your last name**, i.e. replace "adsd" with your last name. Typically, this would be an abbreviated string of the device manufacturer's name.

```c
/*
 * Define the compatible property used for matching devices to this driver,
 * then add our device id structure to the kernel's device table. For a device
 * to be matched with this driver, its device tree node must use the same
 * compatible string as defined here.
 */
static const struct of_device_id led_array_of_match[] = {
        { .compatible = "adsd,led-array", },
        { }
};
MODULE_DEVICE_TABLE(of, led_array_of_match);
```

Listing 4: This code defines what devices the driver is compatible with and adds our module to the kernel's device table. `struct of_device_id` is defined in include/linux/mod_devicetable.h. We only need to set the `compatible` field, which is what's used to match against the `compatible` property in the device tree. `compatible` is of the form "manufacturer,device". `MODULE_DEVICE_TABLE` allows the kernel to load our module if a compatible device is found; in our case, this won't happen because our module is out-of-tree.

Create and test the driver by following these steps:

1. Add the code in Listings 2, 3, and 4 to a file named `led-array.c`. The `probe` and `remove` functions need to be declared before the `led_array_driver` struct, unless you define function prototypes for the `probe` and `remove` functions.
2. Add the following headers:
    - `#include <linux/module.h>`: basic kernel module definitions
    - `#include <linux/platform_device.h>`: platform driver/device definitions
    - `#include <linux/mod_devicetable.h>`: `of_device_id`, `MODULE_DEVICE_TABLE`
3. Compile your `led-array` module. For creating the Makefile, You can copy one of the AD1939 or TPA613A2 Makefiles from previous assignments to use as a starting point, or you can use the one shown in the Appendix of this document.
4. Copy `led-array.ko` to `/lib/modules` on your DE10 Nano.
5. In your serial console (e.g. PuTTY), load your `led-array` module with `insmod`.
6. Look for your driver's print statements by typing

    ```
    $ dmesg | tail
    ```

What happened when loaded your module? Did your driver print "led_array_probe"? It shouldn't have because Linux doesn't know that an `led-array` device exists; we need to add our `led-array` device to the device tree!

Modifying the Device Tree

When we create new hardware (our custom component in Platform Designer), we need to inform Linux that this hardware exists. We do this by creating a new node in the *device tree*. We will build on our `socfpga_cyclone5_audio_mini` device tree.

Open Platform Designer in Quartus so that you can see the base addresses of the system components. Check that the base address in `socfpga_cyclone5_audio_mini.dts` for the `audiomini_led_controller` node contains the correct address. Correct the address if it needs to be corrected in the device tree source so that it reflects what Platform Designer has assigned it for its base address.

Now add a new device tree node for the `led_array` component. This will be a node contained within the root node of the device tree, i.e. just inside `/{...}` in the device tree. The syntax of the node will be:

```
label: node-name@base_address {
    compatible = "manufacturer,model";
    reg = <base_address span>;
};
```

Let's assume that Platform Designer has given the `led_array` component a base address of `0x20`. Since the component is attached to the lightweight bus, which has a base address of `0xff20000`, we add `0x20` to this value. We use the same span that Platform Designer has given the component. The device tree node (named `led-array`) for this component is thus:

```
de10_led_array: led-array@ff200020 {
    compatible = "adsd,led-array";
    reg = <0xff200020 0x8>;
};
```

Listing 5: `led-array` device tree node.

Change the `compatible` property to `"lastname,led-array"` to match what you did in your driver code. Ensure that `led-array`'s base address and span match what has been assigned by Platform Designer.

Important: The compatible strings need to match *exactly*. Otherwise, the driver will not get bound to the device.

Compile your device tree and copy the dtb to your SD card. Now reboot your SoC FPGA and load your `led-array` driver. When you run `dmesg | tail`, you should now see "led_array_probe" printed; similarly, you should see "led_array_remove" printed when you remove from your driver. That was fun, wasn't it? 🥳 🤓

Setting up a State Container and Accessing Device Memory

Printing messages is fun and all, but we actually want to use our platform device. In this section, we'll take some baby steps towards that goal: we'll set up a state container for the led array, get access to the hardware's memory, and turn the LEDs on/off in the probe/remove functions.

The state container design pattern is common in device drivers. Essentially, the state container is a `struct` that *contains* the *state* of our device, e.g. addresses, values, etc.—anything we want to associate with the device and keep track of. One of the primary reasons for this design pattern is that drivers assume they will be bound to multiple devices (you could instantiate `led_array` multiple times in Platform Designer if you had external LEDs connected up to I/O pins), so we need a separate state for each device. The device's state is passed around to any function that needs to access or modify the state. Listing 6 shows a basic state container for our LED array.

```c
/**
 * struct led_array_dev - Private led array device struct.
 * @reg: Base address of the led array component
 * @value: The led array's current value
 *
 * An led_array_dev struct gets created for each led array in the system.
 */
struct led_array_dev {
        void __iomem *reg;
        u8 value;
};
```

Listing 6: LED array state container. The `led_array_dev` **`struct`** contains a pointer to the LED array's base address, as well as a copy of the LED array's value. The `__iomem` token specifies that the pointer points to I/O memory, which allows the kernel to perform some code-correctness checks[2]. `u8` is a kernel data type for unsigned 8-bit integers.

In order to access and control our platform device, the probe function needs to do three primary things, as shown in Listing 7:

1. Allocate kernel-space memory for our state container (line 11 in listing 7)
2. Request and remap the device's physical memory into the kernel's virtual address space (line 24 in listing 7)
3. Attach our state container to the platform device (line 35 in listing 7)

[2] See this great stackoverflow answer for more details on the `__iomem` token.

```c
static int led_array_probe(struct platform_device *pdev)
{
        struct led_array_dev *priv;

        /*
         * Allocate kernel memory for the led array device and set it to 0.
         * GFP_KERNEL specifies that we are allocating normal kernel RAM;
         * see the kmalloc documentation for more info. The allocated memory
         * is automatically freed when the device is removed.
         */
        priv = devm_kzalloc(&pdev->dev, sizeof(struct led_array_dev),
                            GFP_KERNEL);
        if (!priv) {
                pr_err("Failed to allocate memory\n");
                return -ENOMEM;
        }

        /*
         * Request and remap the device's memory region. Requesting the region
         * make sure nobody else can use that memory. The memory is remapped
         * into the kernel's virtual address space becuase we don't have access
         * to physical memory locations.
         */
        priv->reg = devm_platform_ioremap_resource(pdev, 0);
        if (IS_ERR(priv->reg)) {
                pr_err("Failed to request/remap platform device resource\n");
                return PTR_ERR(priv->reg);
        }

        // Initialize LEDs to on, just for fun.
        priv->value = 0xff;
        iowrite32(priv->value, priv->reg);

        // Attach the led array's private data to the platform device's struct.
        platform_set_drvdata(pdev, priv);

        pr_info("led_array_probe successful\n");

        return 0;
}
```

Listing 7: More useful probe function. This probe function allocates kernel memory for the LED array state container, gets a pointer to the LED array's base address, turns the LEDs on, and attaches our state container to the platform device's `driver_data` field.

In the old days (pre-2007 or so), memory and resources that were allocated in `probe()` needed to be manually released in `remove()`, e.g. for every `kmalloc()` used in the `probe` function, there needed be a corresponding `kfree()` in the `remove` function. This led to many bugs during device initialization and detach, which eventually prompted the creation of devres, which stands for (Managed) **Device Resource**. The devres documentation humorously describes the motivation as follows

> As with many other device drivers, libata low level drivers have sufficient bugs in ->remove and ->probe failure path. Well, yes, that's probably because libata low level driver developers are lazy bunch, but aren't all low level driver developers? After spending a day fiddling with braindamaged hardware with no document or braindamaged document, if it's finally working, well, it's working.
>
> So, many low level drivers end up leaking resources on driver detach and having half broken failure path implementation in ->probe() which would leak resources or even cause oops when failure occurs

The upshot of using devres is that resources managed by it are automatically released upon device detach, which eliminates a lot of bugs and makes our lives and code much simpler. devres functions are prefixed with `devm_`. devres is "basically [a] linked list of arbitrarily sized memory areas associated with a `struct device`".

`devm_kzalloc` is the first devres function we use. We use it to allocate memory for our state container. In summary, it creates a memory region of size `sizeof(struct led_array_dev)` (the size of our state container) and adds the corresponding pointer to the `device struct` contained in our platform device (`&pdev->dev` is a pointer to the `device struct`). See the `devm_kzalloc` and `devm_kmalloc` definitions for details.

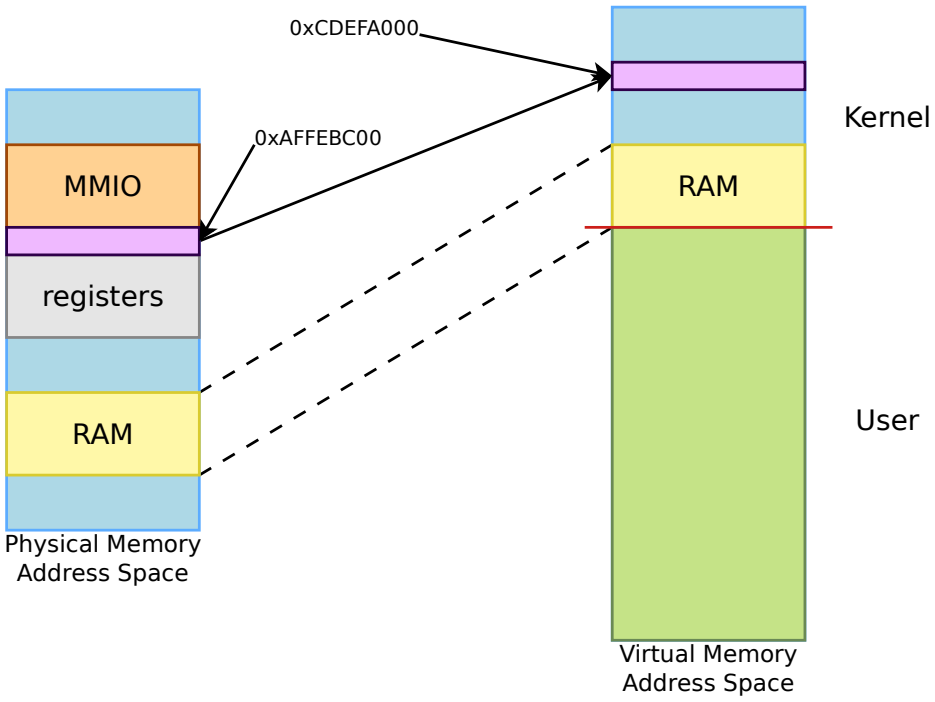

Figure 3: Example of remapping memory-mapped I/O into the kernel's virtual address space. The physical memory region's base address is 0xAFFEBC00, which gets remapped to address 0xCDEFA000 in the kernel's virtual address space. This ioremap diagram was created by Michael Opdenacker from Bootlin and is licensed under CC BY-SA 3.0.

The next thing our probe function has to do is gain access to the LED array's physical memory. This is done with the devm_platform_ioremap_resource function. This function first requests access to the memory region, which ensures that nobody else can use that memory. The function then remaps the physical memory into the kernel's virtual memory space; this remapping is necessary because the kernel doesn't have direct access to the device's physical address space; Figure 3 shows an example of memory remapping. The output of devm_platform_ioremap_resource is a pointer (in virtual address space) to the LED array's base address.

Line 32 in listing 7 is used to turn the LEDs on. iowrite32 takes a 32-bit value and a **void** pointer, then handles all the architecture-dependent details needed to actually write to our hardware (writing to memory-mapped I/O is different on x86 vs. ARM vs. PowerPC vs. MIPS, etc.).

The last thing our probe function needs to do is attach our state container to it's associated device **struct**. platform_set_drvdata() handles this for us. It takes a pointer to a platform device and a pointer to our state container, and then sets the device's driver_data field, i.e. pdev->dev->driver_data = priv. We need to attach our state container to the device because the (platform) device is what gets passed around to our functions (we've seen this already in our probe/remove functions).

Although there aren't many lines of code, there's a lot going on in the probe function! Fortunately, since we used devres, our remove function is much simpler. All we need to do is get our state container from the platform device struct so we can turn off the LEDS, as shown in listing 8.

```
static int led_array_remove(struct platform_device *pdev)
{
        // Get the led array's private data from the platform device.
        struct led_array_dev *priv = platform_get_drvdata(pdev);

        // Turn the lEDs off, just for kicks.
        iowrite32(0x00, priv->reg);

        pr_info("led_array_remove successful\n");

        return 0;
}
```

Listing 8: More useful remove function. The function gets the LED array's state container from the platform device, then turns the LEDs off. All memory that was allocated and mapped in our probe function is automatically cleaned up by the kernel because we used managed devres functions.

To put this all together, update your driver as follows:

- Add the following headers to your file:
  - #include <linux/io.h>: iowrite32/ioread32 functions
  - #include <linux/types.h>: u8 type defintion
- Add the code from listing 6 to the top of your file, below the include statements.
- Add the probe and remove functions from listings 7 and 8; these need to be above your **static struct platform_driver** led_array_driver definition, unless you use function prototypes.

Compile the driver and copy it to the /lib/modules on your DE10 Nano. Loading your driver should turn all 8 LEDs on, and removing it should turn the LEDs off! 🐔

Miscdevice Creation

We're making progress! We've now turned LEDs on in the most complicated way yet! But wouldn't it be nice to control the LEDs outside of the probe function? To do this, we'll create a *character device* using the misc subsystem. This subsystem is just a light wrapper around the character device API, but it saves a lot of headache and boilerplate.

First, we need to add a `miscdevice` to our state container, as shown in listing 9. By putting the `miscdevice` in our state container, each LED array in our system will get it's own `miscdevice`; although we only have one LED array, designing our driver to handle multiple LED arrays is the best practice. Listing 9 also contains a *mutex*, but we can ignore that until we implement the character device's write function.

```
/**
 * struct led_array_dev - Private led array device struct.
 * @miscdev: miscdevice used to create a char device for the led array
 * @reg: Base address of the led array component
 * @value: The led array's current value
 * @lock: mutex used to prevent concurrent writes to the led array
 *
 * An led_array_dev struct gets created for each led array in the system.
 */
struct led_array_dev {
        struct miscdevice miscdev;
        void __iomem *reg;
        u8 value;
        struct mutex lock;
};
```

Listing 9: Updated LED array state container. `miscdev` will contain the `miscdevice`. `struct mutex lock` will be used to ensure that concurrent writes to the LED array happen in order (i.e. without interruption).

Once we have the `miscdev` in our state container, we can initialize it in our probe function, as highlighted in listing 10. The fields we initialize include the minor number, which we set to be dynamically allocated. **Note:** Devices have **major** and **minor** numbers. The major ID or number identifies the general class of the device, which is used by the kernel to lookup the appropriate driver for this type of device. The minor number identifies a particular device within the class.

Other fields include the character device name that will show up in `/dev/`; the file operations supported by the device; and the device's parent, which in our case is the platform device (the "physical" device, so to speak). See the `miscdevice` definition to see all of the fields. Once the device is initialized, all we need to do is register it with the misc subsystem using `misc_register()`.

All we need to do in the remove function is deregister the `miscdev`, as shown in listing 11.

Using the code in listings 10 and 11, the misc subsystem will create a character device for us a `/dev/led_array`. To actually use the device, we need to define what file operations are supported.

```
// Initialize LEDs to on, just for fun.
priv->value = 0xff;
iowrite32(priv->value, priv->reg);

// Initialize the misc device parameters
priv->miscdev.minor = MISC_DYNAMIC_MINOR;
priv->miscdev.name = "led_array";
priv->miscdev.fops = &led_array_fops;
priv->miscdev.parent = &pdev->dev;

// Register the misc device; this creates a char dev at /dev/led_array
ret = misc_register(&priv->miscdev);
if (ret) {
        pr_err("Failed to register misc device");
        return ret;
}

// Attach the led array's private data to the platform device's struct.
platform_set_drvdata(pdev, priv);
```

Listing 10: How to initialize and register the `miscdev`. The `fops` field is a pointer to our `file_operations` **`struct`**, and the parent field is a pointer to the `device` **`struct`** contained within our `platform_device` **`struct`**. `ret` **needs to be declared as an `int` at the top of the probe function.**

```
struct led_array_dev *priv = platform_get_drvdata(pdev);

// Turn the lEDs off, just for kicks.
iowrite32(0x00, priv->reg);

// Deregister the misc device and remove the /dev/led_array file.
misc_deregister(&priv->miscdev);
```

Listing 11: How to deregister the `miscdev` in our `remove` function.

Read/Write Functions

The `file_operations` **`struct`** is used to define what file operations are supported by a character device. The main operations we need are `open`, `close`, `read`, and `write`. The misc subsystem takes care of `open` and `close` for us, so we just have to define the `read` and `write` functions. Listing 12 shows the file operations structure we need to create.

```
// File operations struct for our miscdev
static const struct file_operations led_array_fops = {
        .owner = THIS_MODULE,
        .read = led_array_read,
        .write = led_array_write,
};
```

Listing 12: LED array `file_operations` **`struct`**. Setting the `owner` field to to `THIS_MODULE` prevents our LED array module from being unloaded while its operations are in use. The `read` and `write` fields are set as pointers to our read and write functions.

The struct itself is pretty simple; all the heavy-lifting happens in the `read` and `write` functions. We'll start with the `read` function shown in listing 13, as it's the simpler of the two. The comments in listing 13 explain what the function does.

```
/**
 * led_array_read() - Read method for the led_array char device
 * @file: Pointer to the char device file struct.
 * @buf: User-space buffer to read the value into.
 * @count: The number of bytes being requested.
 * @offset: The byte offset in the file being read from.
 *
 * On success, the number of bytes read is returned and @offset is advanced
 * by this number. On error, a negative error value is returned.
 */
static ssize_t led_array_read(struct file *file, char __user *buf,
        size_t count, loff_t *offset)
{
        /*
         * Get the device's private data from the file struct's private_data
         * field. The private_data field is equal to the miscdev field in the
         * led_array_dev struct. container_of returns the led_array_dev struct
         * that contains the miscdev in private_data.
         */
        struct led_array_dev *priv = container_of(file->private_data,
                struct led_array_dev, miscdev);

        // Read value from hardware, in case it changed without our knowledge
        priv->value = ioread32(priv->reg);

        /*
         * Copy data to user-space and return the number of bytes copied.
         * Returns an error if the copy did not work.
         */
        return simple_read_from_buffer(buf, count, offset,
                &priv->value, sizeof(priv->value));
}
```

Listing 13: Character device read function. The read function gets our state container, reads the value from LED array value from hardware, then writes the value back to user-space.

The trickiest part is the `container_of` macro. Essentially, `container_of` finds the `struct led_array_dev` that *contains* the `miscdev` pointed to by `file->private_data` (remember that our state container conatins a `miscdev`); the misc subsystem assigns our `miscdev` to `file->private_data`.

`simple_read_from_buffer` handles all of the buffer-copying, error-checking, and pointer-setting that we would have to do to properly copy data from a user-space buffer into kernel-space. For numerous reasons, the kernel can't directly dereference user-space pointers; see this discussion in LDD3 to learn more. When a program reads from a character device, it will read until end-of-file (EOF) is reached; EOF is signified by returning a 0 in our `read` function. However, if we only return 0, the user-space program won't properly receive the data it tried to read. We need to return the number of bytes that were read and advance the file `offset` pointer by that number of bytes. We keep doing this until there are no bytes left to read, at which point we return 0 to signal EOF has been reached. Thankfully, `simple_read_from_buffer()` handles all of this for us.

The operations we have to do in our `write` function are conceptually similar to the `read` function, particularly in how the return value and `offset` value are handled. See the `simple_write_to_buffer` definition and documentation for details.

```
/**
 * led_array_write() - Write method for the led_array char device
 * @file: Pointer to the char device file struct.
 * @buf: User-space buffer to read the value from.
 * @count: The number of bytes being written.
 * @offset: The byte offset in the file being written to.
 *
 * On success, the number of bytes written is returned and the offset @offset is
 * advanced by this number. On error, a negative value is returned.
 */
static ssize_t led_array_write(struct file *file, const char __user *buf,
        size_t count, loff_t *offset)
{
        int ret;
        struct led_array_dev *priv = container_of(file->private_data,
                struct led_array_dev, miscdev);

        if (count > sizeof(priv->value)) {
                pr_notice("Input value is too large\n");
                return -EINVAL;
        }

        mutex_lock(&priv->lock);

        // Copy data from user-space into priv->value.
        ret = simple_write_to_buffer(&priv->value, sizeof(priv->value),
                offset, buf, count);
        if (ret < 0) {
                // simple_write_to_buffer returned a negative error code
                goto unlock;
        }

        iowrite32(priv->value, priv->reg);

unlock:
        mutex_unlock(&priv->lock);
        return ret;
}
```

Listing 14: Character device write function. The write function doesn't allow values larger than 8-bits to be written. A mutex is used to prevent multiple writes to `/dev/led_array` from being reordered.

The new concept in our write function is that we use a *mutex* to protect against race conditions. Mutex is short for *mutual exclusion*, which essentially means that only one thread/process can access a shared resource at any time. See the mutual exclusion Wikipedia page and the kernel's mutex documentation for more information.

In short, the race condition we need to prevent is as follows: *process A* writes a value to `priv->value`, but gets interrupted before executing `iowrite32()`; *process B* then overwrites a different value to `priv->value`,

and also gets interrupted before executing `iowrite32()`; *process A* resumes and executes `iowrite32()`, but instead of writing the value it put into `priv->value`, it writes the value that *process B* was trying to write. By putting a mutex around the buffer-copying and `iowrite32()`, we prevent multiple writers from accessing the *critical section* at a time.

We're now ready to implement our `miscdev`. Update your code as follows:

1. Add the following headers:
    - `#include <linux/mutex.h>`: mutex defintions
    - `#include <linux/miscdevice.h>`: miscdevice defintions
    - `#include <linux/fs.h>`: `simple_read_from_buffer` and `simple_write_to_buffer`
2. Update your state container according to listing 9.
3. Update your probe and remove functions according to listings 10 and 11.
4. Add the `fops` structure from listing 12 above your probe function.
5. Add the read and write functions from listings 13 and 14 above your `fops` structure.

Compile the driver and copy it to `/lib/modules` on your DE10 Nano. Remove the previous module using `rmmod` if you haven't already, then load the new module. You should see a character device show up at `/dev/led_array`.

Test the character device by writing a value to it using `echo`, e.g.:

```
$ echo -n 5 > /dev/led_array
```

The `-n` tells `echo` to not include a trailing newline, which is important because our write function only accepts one-byte values (`5\n` is two bytes).

Read the value you wrote back from the character device as follows:

```
$ awk 1 /dev/led_array
```

We use `awk` instead of `cat` because `cat` assumes that the file ends with a newline, which ours doesn't; if we were to use `cat`, the value would be printed right before the console's prompt, which is very difficult to read (try it yourself).

You'll notice that the values you write to the LEDs don't necessarily make sense, e.g. writing `5` doesn't result in the binary version of `5` being displayed on the LEDs. This is because the write function writes the raw bytes that are passed to it. When we use `echo`, we are writing *ascii characters*; consult an ascii table to figure out how to write specific binary values to the LEDs.

Exporting a Device Attribute with sysfs

Writing to our LEDs via the character device is great, but having to do all those ascii conversions is a pain. We could get around this by writing a program that opens the character device file and writes to it, but a lot of times we just want to use simple command line tools like `echo` and `cat`, just like we did in lab 7. To do this, we need to create a *device attribute* for the LED array value and export it via sysfs. For an overview of sysfs and attributes, see the kernel's sysfs documentation; the kernel's device documentation also contains some information about device attributes. **Read these documentation pages before moving on.**

In short, **sysfs** is a RAM-based filesystem used to export kernel data structures and their attributes to userspace.

Creating a Device Attribute

To simplify the creation of our device attribute, we'll use the `DEVICE_ATTR_RW()` macro; doing so removes most of the boilerplate required to define a read/write attribute. Once we've created a device attribute, we add it to an attribute group so the kernel can export the attribute for us. See the `ATTRIBUTE_GROUPS` macro to for some details on how it creates the attribute group for us. This post by Greg Kroah-Hartman (one of the main Linux kernel maintainers) provides details on how to properly create sysfs files, but the post is a little bit dated and doesn't reflect the interfaces/macros we are using; the post can be regarded as a description of how lower-level kernel code creates sysfs files for us. Listing 15 shows the code we use to create our "value" device attribute.

```
// Declare the "value" attribute as read/write
static DEVICE_ATTR_RW(value);

// Create an atribute group so the device core can export the attributes for us.
static struct attribute *led_array_attrs[] = {
        &dev_attr_value.attr,
        NULL,
};
ATTRIBUTE_GROUPS(led_array);
```

Listing 15: How to create a read/write device attribute. The `DEVICE_ATTR_RW` macro creates the `device_attribute` `struct` for us, resulting in a variable called `dev_attr_value`. We then create an attribute group from a list of attributes. The `ATTRIBUTE_GROUPS()` macro expands `led_array` into `led_array_attrs` (the name of our attribute list).

Once we've created the attribute, we need to define its associated `show` and `store` functions. We'll start with the `show` function, shown in listing 16. The `show` function is responsible for returning the attribute as an ascii text-file; it gets called when anybody in user-space tries to read the attribute. Since our value is a `u8`, we use `scnprintf` to format the value as a string and place it into the user-space buffer. See printk-specifiers for information on the format string used in `scnprintf`.

The `store` method shown in listing 17 is called whenever someone in user-space writes to the attribute. We use `kstrtou8` to parse the string we were given into a `u8`; this function supports conversion from decimal-, hex-, and octal-formatted strings, and it ensures that the parsed value fits into an unsigned 8-bit integer.

```c
/**
 * value_show() - Return the led array value to user-space via sysfs.
 * @dev: Device structure for the led array. This device struct is embedded in
 *       the led array's platform device struct.
 * @attr: The specific sysfs attribute being requested; this is unused since we
 *        only have one attribute.
 * @buf: Buffer that gets returned to the led array value to user-space.
 */
static ssize_t value_show(struct device *dev,
        struct device_attribute *attr, char *buf)
{
        // Get the private led_array data out of the dev struct
        struct led_array_dev *priv = dev_get_drvdata(dev);

        /*
         * the value in reg should generally be equal to the "value" shadow
         * register; the only exception would be when the register was modified
         * directly in hardware or over jtag, thereby bypassing our driver
         */
        priv->value = ioread32(priv->reg);

        /*
         * Put the value in the user-space buffer. The buffer provided by sysfs
         * is always PAGE_SIZE bytes. Return the number of bytes written to buf.
         */
        return scnprintf(buf, PAGE_SIZE, "%u\n", priv->value);
}
```

Listing 16: sysfs show method. This method gets our state container out of device associated with the attribute being read. It then reads the value from hardware, and fills the user-space buffer with a string that contains the LED array value.

```c
/**
 * value_store() - Store the value that was written to the "value" attribute
 * @dev: Device structure for the led array. This device struct is embedded in
 *       the led array's platform device struct.
 * @attr: The specific sysfs attribute being requested; this is unused since we
 *        only have one attribute.
 * @buf: Buffer that contains the led array value being written.
 * @size: The number of bytes being written.
 */
static ssize_t value_store(struct device *dev,
        struct device_attribute *attr, const char *buf, size_t size)
{
        struct led_array_dev *priv = dev_get_drvdata(dev);
        int ret;

        // Parse the string we were passed and put it into a u8, if possible.
        ret = kstrtou8(buf, 0, &priv->value);
        if (ret < 0) {
                // kstrtou8 returned an error
                return ret;
        }

        iowrite32(priv->value, priv->reg);

        // Write was succesful, so we return the number of bytes we wrote.
        ret = size;

        return ret;
}
```

Listing 17: sysfs store method. This is similar to show method in listing 16, except it receives and parses the string written by the user, then stores it in `priv->value`. We don't need a mutex this time because the kernel layer above sysfs handles the mutex's for us; see this stackoverflow post for details.

Attaching Device Attributes to the Platform Device

With the show and store functions defined, we're almost ready to use our attribute. We just need to attach the attribute group to our platform driver so the driver core can create and export the attribute without race conditions. Listing 18 shows how to do this.

```
/**
 * struct led_array_driver - Platform driver struct for the led array driver
 * @probe: Function that's called when a device is found
 * @remove: Function that's called when a device is removed
 * @driver.owner: Which module owns this driver
 * @driver.name: Name of the led array driver
 * @driver.of_match_table: Device tree match table
 * @driver.dev_groups: LED array sysfs attribute group; this
 *                     allows the driver core to create the
 *                     attribute(s) without race conditions.
 */
static struct platform_driver led_array_driver = {
        .probe = led_array_probe,
        .remove = led_array_remove,
        .driver = {
                .owner = THIS_MODULE,
                .name = "led-array",
                .of_match_table = led_array_of_match,
                .dev_groups = led_array_groups,
        },
};
```

Listing 18: How to add our attribute group to our platform driver, using the `dev_groups` field. See the device_driver struct documentation for details.

Now we're ready to test the attribute. **Update your driver as follows**:

1. Add `#include <linux/kernel.h>`: `kstrtou8`
2. Add the code in listings 15, 16, and 17 to your driver; the `show` and `store` functions need to declared before you create the attribute (listing 15).
3. Modify your `led_array_driver` `struct` according to listing 18

Compile the driver and copy it to `/lib/modules` on your DE10 Nano again. Remove any previously-loaded driver and load your new driver.

If everything worked correctly, there should be a `value` file located at `/sys/devices/platform/<base-address>.led_array` where `<base-address>` is the physical address of your LED array. `cd` to that directory and write/read values from/to the "value" attribute, e.g.

```
$ echo 0x83 > value
```

```
$ cat value
```

If all goes well, you should be able to control your LEDs! 🎉 On the command line, using sysfs is much more convenient than using the character device.

Attaching Device Attributes to the Misc Device

Although using sysfs is pretty convenient, the path to our "value" attribute isn't. By adding our device attribute to the `miscdev`, we can make the attribute show up at a slightly more convenient path: `/sys/class/misc/led_array/value`. Listing 19 shows how to do that.

```
// Initialize the misc device parameters
priv->miscdev.minor = MISC_DYNAMIC_MINOR;
priv->miscdev.name = "led_array";
priv->miscdev.fops = &led_array_fops;
priv->miscdev.parent = &pdev->dev;
priv->miscdev.groups = led_array_groups;
```

Listing 19: Adding our attribute group to the `miscdev`. To do this, we simply assign our attribute group to the `groups` field.

Update your driver's probe function according to listing 19. Recompile your driver, copy it to `/lib/modules` on your DE10 Nano, `rmmod` any previous driver, and then load your new driver. You should now be able to control your LEDs with the "value" attribute located at `/sys/class/misc/led_array/value`.

Phew! There are a lot of ways to write to our LEDs!

Here are locations that the led array device and associated driver show up in Linux (Your address is likely to be different depending on how Platform Designer assigned the base address.):

- `/dev/led_array` (raw bytes)
- `/sys/devices/platform/ff200020.led_array/value` (sysfs device attribute tied to the platform_driver struct)
- `/sys/bus/platform/devices/ff200020.led_array/value` (basically the same as the directory above, but just in a different place)
- `/sys/class/misc/led_array/value` (sysfs device attribute tied to the miscdevice struct)
- `/sys/devices/virtual/misc/led_array/value` (same as above directory)

Information about the driver itself is in

- `/sys/bus/platform/drivers/led-array` This directory has symlink(s) to the device(s) that are bound to it (/sys/devices/platform/...) It also has a symlink to kernel module information (/sys/module/led_array).
- `/sys/` has a lot of information, and a lot of symlinks between the different parts of the tree (e.g. bus, devices, class, etc.). For example, the led_array character device shows up as a symlink in /sys/dev/char/, which links to /sys/devices/virtual/misc/led_array.
- `/proc/device-tree` This shows the live device tree.

LED Patterns Revisited

All we ever do is light up LEDs, so let's do light 'em up again! For fun, we're going to create LED patterns again, but this time we'll use our device driver and *shell scripts*!

We'll learn by example. Listing 20 shows an example that creates an "alternating" pattern. Listing 21 left-shifts an arbitrary `LED_VAL`. See this bash cheatsheet for more details on bash syntax.

Both scripts start with `#!/bin/bash`. This is known as a "shebang". It tells the interpreter that called the script (bash, in our case) which interpreter to use when executing the rest of the script. This may seem a little weird when we are using a bash terminal to call a bash script, but we can also use shebangs to execute python scripts, perl scripts, etc. The main upside of using a shebang is that it allows us to execute the script using `./`, e.g. `./led_pattern.sh`, instead of `bash led_pattern.sh`; in order to execute the script this way, the script file needs to have execute permissions.

```
#!/bin/bash
LED_ARRAY=/sys/devices/platform/ff200010.led_array/value

while true
do
    echo 0x55 > $LED_ARRAY
    sleep 0.25
    echo 0xaa > $LED_ARRAY
    sleep 0.25
done
```

Listing 20: Shell script to create an "alternating" LED pattern. `sleep` takes arguments in *seconds*. `LED_ARRAY` is a path to your "value" attribute; your path might differ from what's shown.

```
#!/bin/bash
LED_ARRAY=/sys/devices/platform/ff200010.led_array/value
LED_VAL=0x53;

while true
do
    echo $LED_VAL > $LED_ARRAY
    sleep 0.15
    # left-shift LED_VAL; wrap the value when it overflows 0xff
    LED_VAL=$((LED_VAL << 1) % 0xff))
done
```

Listing 21: Shell script to left-shift an arbitrary value on the LEDs.cd Anything inside `$(())` is treated as a math expression.

Test out the scripts in listings 20 and 21:

1. Put the code in listing 20 into a file called `led_pattern0.sh`
2. Put the code in listing 21 into a file called `led_pattern1.sh`
3. Copy the shell files over to your DE10 Nano's filesystem.
4. In PuTTY, give the shell scripts execute permissions with `chmod +x`
5. In PuTTY, execute the scripts, e.g. `./led_pattern0.sh`; the scripts are infinite loops, so you will need to use ctrl+c to stop the script.

Demo

1. Using the character device at `/dev/led_array`, make the binary value `0b00101011` show up on your LEDs.
2. Create a shell script that displays a pattern of your choosing on the LEDs; the pattern **can't** be the exact same as those in listings 20 and 21.
3. Prove that you wrote your driver by showing the output of

   ```
   $ modinfo led-array.ko
   ```

   Your name should show up as the author.

Submission

- Make sure you have committed all the files you created or modified during this lab. Commit your updated audio mini device tree to your `linux-socfpga` repository.
- Create a pull request from the `main` branch into the `feedback` branch.
  - @mention me in the comments.
  - Include a URL/link to your audio mini device tree in your `linux-socfpga` repository.

Appendix

Compiling Kernel Modules

Kernel modules must be built with the kernel's build system, known as kbuild. Modules can either be built in-tree (within the kernel's source tree) or out-of-tree (externally). We will be building our modules out-of-tree.

To build external modules, we need a copy of the kernel configuration and header files for the kernel we are building for. Since we already have cloned the Linux kernel repository, we can compile against that repository.

> **Note:** Kernel modules must be compiled against the kernel version you are using; doing otherwise will result in the module not loading.

kbuild uses a special build file syntax, described in the Linux Kernel Makefiles documentation. All we need to be concerned with right now is what are known as "goal definitions". For building a module that only has a single source file, our goal definition will look like this: `obj-m := <module-name>.o`, where `<module-name>` is the name of your source file (without an extension).

If we were building in-tree, all we would need is that single goal definition. But since we are building out of tree, we need to tell `make` where the kernel source is and where our module is:

```
$ make -C <kernel-source-path> M=<module-path>
```

Rather than typing this command out all the time, kernel developers create a Makefile like shown in Listing 22.

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := led-array.o

else
# normal makefile

# path to kernel directory
KDIR ?= <path_to_kernel_repository>

default:
    $(MAKE) -C $(KDIR) ARCH=arm M=$$PWD CROSS_COMPILE=arm-linux-gnueabihf-

clean:
    $(MAKE) -C $(KDIR) ARCH=arm M=$$PWD clean
endif
```

Listing 22: Example Makefile for cross-compiling an external module

The Makefile in Listing 22 is invoked by just running `make`. When you are out-of-tree, the Makefile will call the appropriate `make` command to build the module using kbuild, which will make another pass through the Makefile, this time using the kbuild part of the Makefile.

After compiling a module, a `<module-name>.ko` file will be created; this is the kernel module artifact that we will load/unload.

For more information on building external modules, refer to this section of the Kbuild documentation.

Loading/Unloading Kernel Modules

We will use `insmod` to insert kernel modules, and `rmmod` to remove kernel modules. `modprobe` is a more robust program, but it only looks for modules in `/lib/modules/`uname -r``, which is not necessarily where our modules will be; we could use `modprobe`, but we will opt for the simpler `insmod`/`rmmod`.